Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado de Matemáticas e Ingeniería Informática

Trabajo Fin de Grado

HERRAMIENTA SOFTWARE PARA RECONOCIMIENTO DE FLORES A PARTIR DE IMÁGENES

Autor: Alfonso Castro Gil
Tutor: Luis Fernando Lago Fernández

JUNIO 2018

HERRAMIENTA SOFTWARE PARA RECONOCIMIENTO DE FLORES A PARTIR DE IMÁGENES

Autor: Alfonso Castro Gil
Tutor: Luis Fernando Lago Fernández

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JUNIO 2018

Resumen

Resumen

El reconocimiento de imágenes es un campo muy utilizado en la actualidad, ya que resulta bastante útil en la realización de diversas tareas, las cuales en ocasiones resultan complejas para el ojo humano.

Las técnicas más utilizadas han sido las Redes Neuronales, las cuales simulan las neuronas de una persona o animal y dada una entrada de datos, imprimen una salida proveniente de la realización de operaciones computacionales. Con el paso de los años, se ha ido evolucionando y las Redes Neuronales Convolucionales son las más empleadas. Éstas son similares a las anteriores pero de estructura más compleja y resultados más eficientes.

En este proyecto experimental el objetivo principal consiste en la implementación de un clasificador que sea capaz de identificar una flor a partir de una imagen. Para ello se han utilizado las Redes Neuronales Convolucionales y un conjunto de bases de datos que han permitido realizar todas las pruebas pertinentes para conseguir el resultado de eficiencia deseado.

Palabras Clave

Redes Neuronales, Redes Convolucionales, Reconocimiento de Imágenes, Flores

Abstract

Nowadays, images identification is a widely used field (area) given that it is very useful to the realization of diverse tasks, which in some ocasions, are complex for the human eye.

The most used techniques are Neural Networks that human or animalŠs neurons and gives an output resulting from computational operations. Over the years it has evoluted and the Convolutional Neural Networks are the most significant in this área. These are similar to the others but they are more complex and efficient.

In this experimental proyect the principal aim consists on the implementation of a classifier able to identify flowers from one image. To get it, Convolutional Neural Networks and databases have been used to carry out all pertinent trials to reach efficient results.

Key words

Neuronal Networks, Convolutional Neural Networks, Image Recongnition, Flowers

Agradecimientos

A todos mis profesores de Matemáticas, ya que sin ellos nunca habría descubierto el amor por las Matemáticas. A mis compañeros y amigos del Doble Grado, por ser mi segunda familia durante estos 5 años. A mi gran amigo Guille, por ser tan o más loco que yo con las Matemáticas. A mi amigo-hermano Aitor, por esa amistad eterna. A mis amigos de Coslada que han permanecido a mi lado siempre.

A mi tutor Luis, por su actitud, su ánimo y su forma de plantear la realización del TFG ha sido perfecta.

A mi pareja María, por el apoyo, el ánimo y el cariño continuo durante todo el proceso de mi realización de este TFG. A mi tía Silvia, por ser la persona que fue mi referente para estudiar Informática y ser mi segunda madre. A mis abuelos Alfonso y María Esperanza por cuidarme y darme todo su cariño desde que nací. A mi madre María, por la constancia, el apoyo y el cariño incondicional desde que tengo uso de razón. A mi hermano Javier, por ser mi mejor amigo y estar siempre a mi lado.

Y, por último, a mi padre Ángel, por ser la persona que me dio esta maravillosa idea y la motivación necesaria para realizar este Trabajo de Fin de Grado. A ti va dedicado especialmente con todo mi cariño.

A todos los mencionados y muchos más que faltan que han colaborado en el camino a mi objetivo, MUCHAS GRACIAS.

Índice general

Índice de Figuras

Índice de Tablas

1

Introducción

1.1. Motivación del proyecto

La finalidad de este Trabajo de Fin de Grado (TFG) es la realización de una herramienta software para la clasificación de imágenes de flores. Dicha herramienta se realizará mediante el uso de Redes Neuronales Artificiales (RNA), más precisamente, de Redes Convolucionales.

Las RNA imitan el funcionamiento de las redes neuronales de los organismos vivos: un conjunto de neuronas conectadas entre sí y que trabajan en conjunto, sin que haya una tarea concreta para cada una. Con la experiencia, las neuronas van creando y reforzando sus conexiones para aprender nuevas ideas.

Actualmente, es bastante importante la clasificación y etiquetado de imágenes de manera automática, ya que facilita a muchas personas en su ámbito profesional la realización de aquellas tareas relacionadas con etiquetado de imágenes, aparte del avance tecnológico en la superación de la máquina al ser humano. Además, el hecho de que los objetos a clasificar sean flores nos da una dificultad más, ya que existen muchos ejemplares de flores muy parecidos entre sí, como se puede apreciar en la siguiente figura 1.1.

Figura 1.1: Ejemplo de flores similares entre si [1].

Para esta tarea, un recurso bastante eficaz será el uso de redes neuronales convolucionales (CNN), las cuales tienen actualmente mucho éxito ya que han sido creadas principalmente para ese propósito y sus resultados son muy destacados. Se nos plantea la dificultad de la escasa existencia de bases de datos (BBDD) de flores, por lo que tendremos que encontrar la base datos más amplia posible y que nos permita desarrolar una herramienta software eficaz. Además, dicha herramienta podrá utilizarse con uso recreativo mediante su adaptación a una aplicación móvil que pueda ser utilizada en excursiones o viajes con el fin de averiguar la identidad de una flor.

1.2. Objetivos

Los objetivos que queremos conseguir en este Trabajo de Fin de Grado son los siguientes:

1. Realizar el estudio y la formación con respecto a las Redes Neuronales (especialmente las Redes Convolucionales).

2. Realizar el estudio y la formación con respecto al uso de Tensorflow.

3. Encontrar y usar una base de datos de flores amplia.

4. Desarrollar un clasificador de flores que sea capaz de clasificar cualquier imagen de una flor con la mayor eficacia posible, en comparación a la eficacia de las redes existentes con las bases de datos que vamos a utilizar. Actualmente, los clasificadores de imágenes de flores existentes con las bases de datos que utilizaremos consiguen una eficacia del 80-90 %, por lo que nosotros buscaremos encontrarnos en ese intervalo o, incluso, superarlo.

1.3. Metodología y plan de trabajo

Para la resolución y obtención de todos los objetivos hemos seguido los siguientes procedimientos:

1. Búsqueda bibliográfica y estudio del arte en relación con la clasificación de flores, las RNA y, específicamente, las CNN, y la herramienta *Tensorflow* y su aportación a las CNN. Además de la bibliografía, se han utilizado recursos online como los cursos de Stanford de CNN [13] y los tutoriales de *Tensorflow* [14].
2. Búsqueda de BBDD de flores. Finalmente utilizamos *Oxford 17 Flowers* [3] y *Oxford 102 Flowers* [2].
3. Realización de pruebas con diferentes arquitecturas de redes neuronales y búsqueda de redes preentrenadas hasta encontrar el clasificador óptimo.

2

Reconocimiento de imágenes. Estado del arte

2.1. Clasificación de imágenes.

2.1.1. Introducción

En los últimos 50 años, la clasificación de imágenes ha sido uno de los puntos más trabajados y que más se ha investigado al respecto, para poder conseguir resultados asombrosos.

En el 1954, aparecieron las técnicas basadas en puntos característicos, también conocidas como *Bag of words* (BoW) [15]. Éstas técnicas estaban basadas en encontrar ciertos puntos clave, como puede ser una linea vertical u horizontal, o una esquina, y realizar la cuenta del número de apariciones que tenían en la imagen y guardar esos resultados en un vector. Era una simulación al algoritmo de búsqueda de las palabras más frecuentes en un texto, que consistía en contar el número de veces que aparecía una palabra en un texto cualquiera.

En la década de los años 50, aparecieron las Redes Neuronales Artificiales (RNA), las cuales similaban el funcionamiento de las neurones de los animales y personas, para ser usadas en el campo de reconocimiento de imágenes. Como entrada utilizaban un vector formado por los píxeles de la imagen, lo cual suponía la pérdida de la relación espacial entre ellos, y después le aplicaban un conjunto de operaciones computacionales para obtener un vector de salida. Esto lo explicamos más extensamente en el apartado 2.2.

Y finalmente, en la década de los 90, aparecieron las Redes Artificiales Convolucionales (CNN), las cuales revolucionarían el mundo de la clasificación de imágenes, ya que añadieron la mejora de considerar una matriz de pixeles como entrada, permitiendo asi la relación espacial entre ellos, y la novedad de la capa Convolucional, la cual le daría su nombre a este tipo de redes.

Otro tipo de redes que aparecieron de manera simultánea a las CNN fueron las Redes Recurrentes, las cuales tienen la característica de estar formadas por enlaces de retroalimentación entre todos los componentes que la forman.

A continuación nos centramos en explicar más detalladamente la técnica de puntos característicos y las Redes Convolucionales.

2.1.2. Técnicas basadas en puntos característicos

Una de las técnicas utilizadas para el reconocimiento de imágenes han sido aquellas basadas en puntos característicos. Éstas consisten, como su propio nombre indica, en la clasificación de una imagen a través de la extración de ciertos puntos característicos, como podría ser una línea vertical, una línea horizontal, una esquina,..., es decir, lo que serían sus características más básicas.

Entre las técnicas existentes, las más destacadas son **SIFT**, **SURF** y **ORB**.

Por un lado, SIFT [16] es un método propuesto por Lowe en 1999, el cual permite la detección de puntos característicos en una imagen. Este método funciona de tal manera que tanto la localización como la descripción no cambia aunque sufra diferentes orientaciones, posiciones o escalas.

Por otro lado, SURF [17], que está basado en su predecesor SIFT, presentado por Herbert Bay en 2006, es un también un método de detección de puntos característicos que asegura que los puntos característicos no varían en función de la escala.

Y, por último ORB [18], que creado por Ethan Rublee, Vicent Rabaud, Kurt Konolige y Gary R. Bradski en 2011, fue presentado como la alternativa a SIFT y SURF para la detección de puntos característicos, basándose en la unión de un detector de puntos clave (FAST [19]) y un descriptor (BRIEF [20]).

Pero con todas estas técnicas se pueden presentar ciertas dificultades en el proceso, como la existencia de translaciones, rotaciones y deformaciones de las imágenes que también las heredan los puntos característicos encontrados, la inexistencia de relación entre puntos característicos o el trabajar con una base de datos muy grande.

Por lo que, en los años 90, aparecieron las primeras redes neuronales convolucionales, las cuales conseguirían revolucionar el mundo de la inteligencia artificial al conseguir resultados fantásticos e incluyendo la novedad de analizar la imagen en forma de matriz, es decir, manteniendo la posición y la relación de los pixeles.

2.1.3. Técnicas actuales basadas en redes neuronales convolucionales

En la actualidad, se han conseguido muchísimos avances de las CNN, y gracias a eso pueden ser utilizadas en muchos entornos y ayudan a la realización de tareas del ser humano.

Por ejemplo, nos permiten realizar la **Detección** de figuras en imágenes [21], indicando si se trata de una persona o un animal. Esto se puede conseguir con diferentes técnicas de **Segmentación** [22], las cuales consisten en dividir la imagen en subzonas en función de su escala de grises o RGB. Así, una vez obtenidos todos los componentes que forman la imagen, se puede llegar a realizar la **Descripción** de la imagen con mayor o menor exactitud.

Otro ejemplo interesante sería el uso de redes neuronales en la conducción, permitiendo la **Autoconducción** de automóviles . Esta funcionalidad se conseigue mediante la detección e identificación de peatones y el resto de automóviles en la vía. Las CNN incluso pueden intuir la **Posición** del cuerpo de una persona [23], analizando la colocación de sus brazos y su tronco y así poder sugerirle o corregirle la postura al conducir.

También es destacable la apliación de CNN en el ámbito naval (especialmente el militar), realizando la **Localización** de barcos, misiles y submarinos, y, sobre todo, es muy importante su aplicación en el el mundo sanitario. Aquí, su funcionalidad principal es ayudar en el **Análisis** de manchas sospechosas en radiografías o ecografías, y así ver si se tratan de manchas malignas o benignas.

Toda la estructura y funcionamiento de las CNN será desarrollado en más adelante en el apartado 2.2.

2.1.4. Bases de datos y benchmarks habituales

A día de hoy existen muchas bases de datos en la red para realizar cualquier estudio que necesitemos, pero entre todas las más destacadas son **MNIST**, **CIFAR-10** e **ImageNet**.

Por un lado, *MNIST* [24] es una base de datos formada por imágenes correspondientes a cifras desde el 0 hasta el 9, escritas a mano cada una de ellas por personas totalmente diferentes. A día de hoy, esta base consta de un total de 70.000 imágenes, repartidas en 60.000 imágenes para el conjunto de entrenamiento y 10.000 imágenes para el conjunto de test. Dichas imágenes han sido sometidas a un preprocesamiento el cual consiste en una normalización de su tamaño y su centralización. Normalmente es la primera base de datos que se suele utilizar para familiarizarse y realizar estudios sencillos, dada su simplicidad con respecto al número de clases que tiene (10 clases) y su tamaño ya que no es muy grande.

Por otro lado, *CIFAR-10* [25] es una base de datos bastante conocida y usada por la mayoría de aprendices del reconocimiento de imágenes. Dicha base consta de 60.000 imágenes en color, las cuales tienen un tamaño de 32 por 32 repartidas en 10 clases (avión, automóvil, ave, gato, ciervo, perro, rana, caballo, barco y camión), por lo que hay 6.000 imágenes por clase. En este caso, utilizamos 50.000 imágenes para el conjunto de entrenamiento (5.000 por clase) y 10.000 imágenes para el conjunto de test. Esta base de datos se utiliza sobre todo en los cursos y universidades para el aprendizaje de la técnica de clasificación de imágenes ya que, como en el caso de MNIST, es una base de datos sencilla y no muy grande.

Y, por último, la base de datos más conocida y más usada por la mayoría de la gente es *ImageNet* [26]. Esto se debe a que es el marco de referencia habitual para probar los diferentes algoritmos. Un ejemplo es la realización de una competición anual llamada ILSVRC ("*ImageNet Large Scale Visual Recognition Challenge*") [27], la cual consiste en la realización de la arquitectura de redes convolucionales que consiga la mayor eficacia posible clasificando dicha base de datos.

ImageNet es una base de datos tan inmensa, que algunos componentes del departamento de Informática de la Universidad de Princeton [28] la definen como: *"A large-scale ontology of images"*. Esto se debe a que está formada por más de cinco mil clases y supera el abrumador número de 14 millones de imágenes en su base de datos [29].

2.1.5. Bases de datos de flores

Si nos centramos en las bases de datos de imágenes de flores, que son realmente las que nos van a ser útiles para nuestro Trabajo de Fin de Grado, podemos destacar, por un lado, las bases de datos de *Oxford 17 Flowers* [3] y *Oxford 102 Flowers* [2]; y por otro lado, la base de datos que aporta *Tensorflow* para realizar las pruebas de redes preentrenadas [30].

Empezando por la última, es la base más simple de las tres mencionadas (aunque no la más pequeña), ya que consiste en una base de datos de 5 clases pero aporta un total de entre 600 y 900 imágenes por clase, llegando a un total de 3630 imágenes. De este modo, consiguen asi un gran número de imágenes por clase para entrenar y obtener resultados eficaces.

Por otro lado, centrándonos ahora en las dos BBDD de *Oxford*, ambas bases de datos están formadas por imágenes de flores, comunmente localizadas en Reino Unido, además de que son las bases de datos de flores más utilizadas en el entorno de reconocimiento de imágenes de flores.

Dichas imágenes son de tamaño grande y escala variable, y ofrecen multitud de variaciones con respecto a la posición, la luz y los componentes incluidos en las imágenes aparte de las flores.

La principal diferencia entre ambas bases de datos es el número de clases que contiene cada una de ellas. *Oxford 17 Flowers*, como su propio nombre indica, tiene un total de 17 clases de flores diferentes, mientras que *Oxford 102 Flowers* posee 102 categorías de flores diferentes, con lo cual ya deja entrever que será una base de datos más amplia. Otra diferencia a destacar entre ambas es el número de imágenes por cada clase. En el primer caso hay un número fijo de 80 imágenes por clase, lo que hace un total de 1360 imágenes en la base de datos, y en el segundo caso hay un número de imágenes que varía entre 40 y 256 por clase, que hacen un total de 8159 imágenes en dicha base de datos.

A continuación, podemos ver una muestra de ambas BBDD en la figura 2.1 y la figura 2.2.

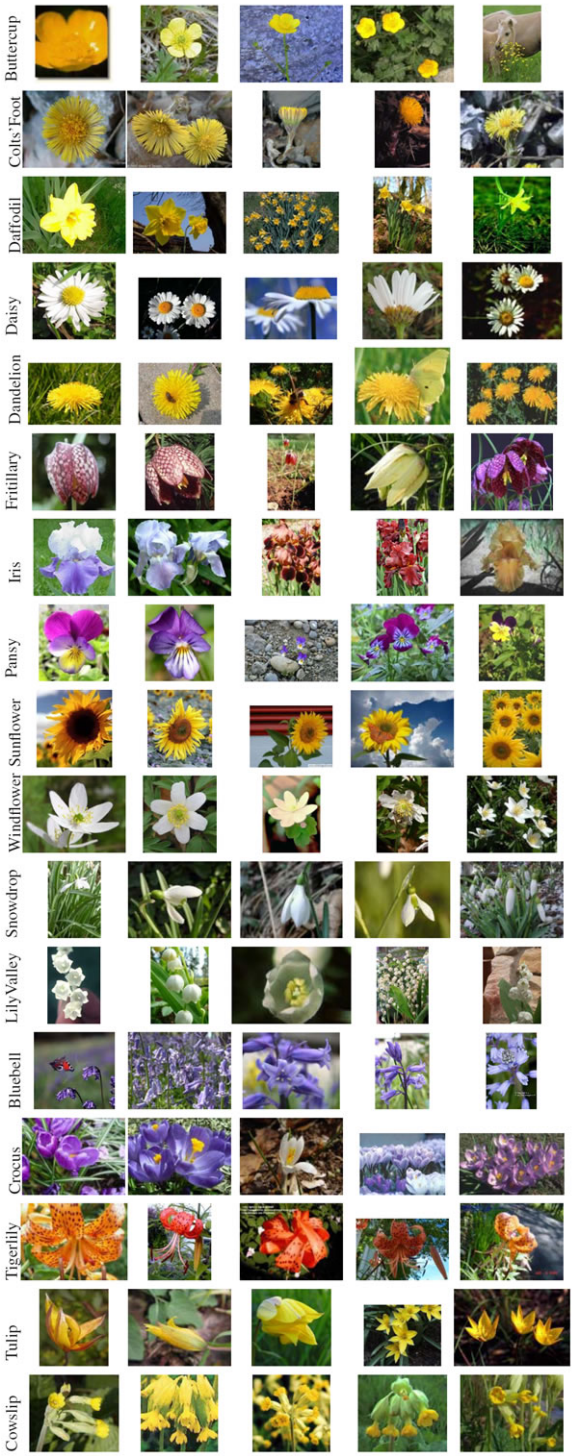

Figura 2.1: Muestra de las categorías de *Oxford 17 Flowers* [3].

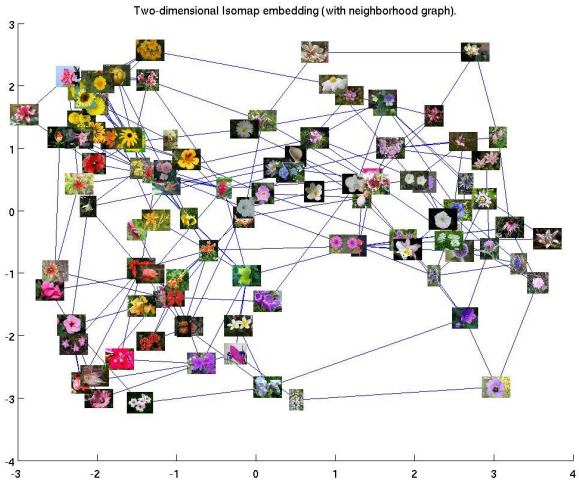

Figura 2.2: Muestra de un grafo con todas las categorías de *Oxford 102 Flowers* [2].

En este TFG vamos a usar las bases de datos de *Oxford*, por lo que nuestro objetivo será conseguir, comparándolo con otros autores que han utilizado dichas bases de datos [31], una eficacia entorno al 80-90 % en test.

2.2. Redes neuronales

2.2.1. Introducción

A día de hoy existen muchos estudios para conocer a ciencia cierta de qué manera funciona el cerebro del ser humano y poder imitar su comportamiento.

Para ello, se utilizan las redes neuronales, las cuales consisten en un conjunto de neuronas conectadas entre sí repartidas en 1 o más capas. Cada capa neuronal tiene asociada un bias, una matriz de pesos y una función de activación propios. El ejemplo más representativo es una red básica *Fully Connected* (FC).

Esta red consiste en un vector de entrada x, un capa neuronal intermedia que la denominaremos h; un vector de salida y; dos matrices de pesos W_1 y W_2 y dos vectores de bias b_1 y b_2 para h e y, respectivamente; y, finalmente, una función de activación para calcular h y otra función de activación para calcular y (puede ser la misma en ambos casos).

Su funcionamiento es sencillo: primero, para calcular el valor de valor de h, realizamos la siguiente operación: $h(x) = \sigma(W_1 x + b_1)$, siendo σ la función de activación para la capa h. Y el siguiente y último paso aplicamos la misma metodología que en el paso anterior pero con sus matrices, vectores y función de activación respectivos: $y(x) = \sigma(W_2 x + b_2)$. En la figura 2.3 podemos verlo más claro visualmente.

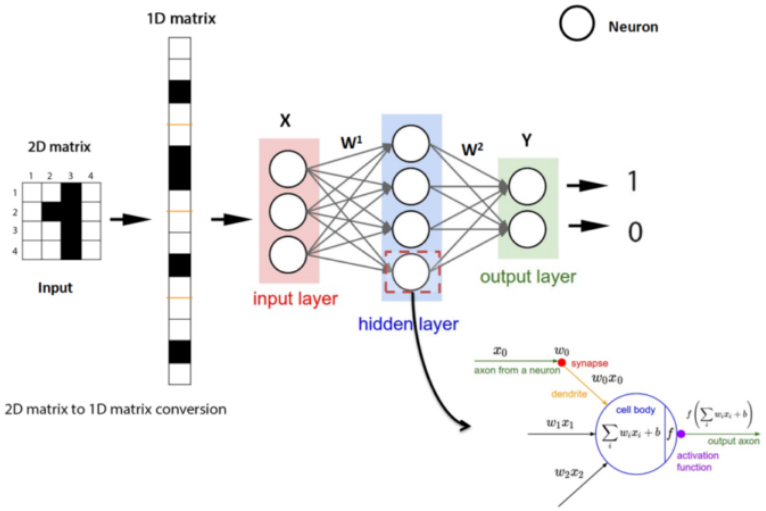

Figura 2.3: Estructura y funcionamiento de una red básica FC [4].

Así finalmente tendríamos nuestro vector de salida y, el cual lo compararíamos con el vector de salida esperado, el cual denominaremos $\hat{y}$, y obtendríamos el error o coste, es decir, una forma de ver cuánto ha errado la red a la hora de calcular la salida. Este coste nos es útil ya no por saber en qué medida nos estamos equivocando al clasificar, sino también para actualizar el valor de las matrices de pesos y los bias. Con esta finalidad, el recurso más conocido y principalmente utilizado es el Descenso por Gradiente (GD), el cual consiste en la actualización de los pesos y bias a partir del gradiente de los pesos y los bias de la función de coste y una constante denominada Constante de Aprendizaje (α) ($W = W - \alpha \cdot \bigtriangledown_W C$, donde C es la función de coste).

Existen multitud de técnicas y métodos de optimización (como Adam o el gradiente por momento de Nesterov) basadas en GD. Todas se basan en la **Regla de Retropropagación**. Ésta consiste en obtener el error de la salida en comparación con la salida que deberíamos obtener y propagar el error hacia atrás, pasando por todas las capas pero en orden inverso y actualizando dichos pesos y bias siguiendo el algoritmo SGD para que en la próxima pasada hacia delante el error disminuya. En la figura 2.4 podemos ver una representación del funcionamiento de la Regla de Retropropagación.

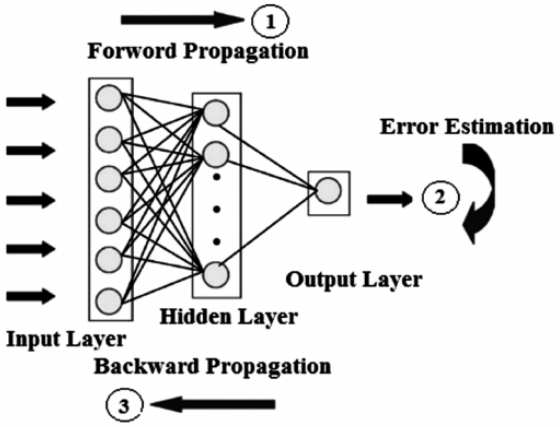

Figura 2.4: Esquema de funcionamiento de la regla de Retropropagación [5].

En la actualidad, se utilizan diferentes arquitecturas de redes neuronales en función del tipo

de problema a resolver. En particular, para nuestro problema que trata con imágenes, será idóneo el uso de las Redes Convolucionales (CNN).

2.2.2. Redes convolucionales

Las redes convolucionales son muy parecidas a las redes neuronales que antes mencionábamos, ya que éstas también están formadas por neuronas con sus pesos y bias respectivos, tienen una entrada y sacan una salida realizando las operaciones pertinentes usando la función de activación que tiene cada neurona. Además, calcularemos el error de la salida obtenida a partir de la salida deseada.

La principal diferencia entre las redes convolucionales y las redes neuronales convencionales consiste en que las primeras suponen que las entradas son imágenes explícitas, por lo que eso nos permite reducir el número de parámetros a usar en la red (pesos y bias) y aumentar la eficacia de la red en comparación con la anterior, y además se permite mantener la relación espacial entre los pixeles de la imagen de entrada. Esto no pasaba antes, ya que se realizaba una conversión de la imagen a un vector y así, se perdía toda la relación espacial. Así que las CNN han supuesto un gran salto en este aspecto.

La estructura de una red convolucional se puede formar a partir de tres capas diferentes: distinguimos **la capa Convolucional**, **la capa de Reducción o *Pooling*** y **la capa *Fully Connected (FC)***.

La red convolucional se carateriza principalmente por la existencia de la capa Convolucional, que le da el nombre a la propia red. Esta capa toma como entrada una matriz y aplica un filtro de nxn dimensiones a través de toda la imagen para obtener una nueva matriz de menor tamaño.

Esto se consigue a través de la aplicación de la çonvolución", un filtro de tamaño nxn que va recorriendo toda la matriz y va realizando el producto escalar de todas las posiciones que están dentro del filtro con cada una de las posiciones de la imagen, para finalmente conseguir el contenido de cada una de las posiciones, aplicando la función de activación asociada (normalmente una ReLU), con tamaño 1x1 de la nueva matriz de salida. Este procedimiento lo podemos ver en la figura 2.5.

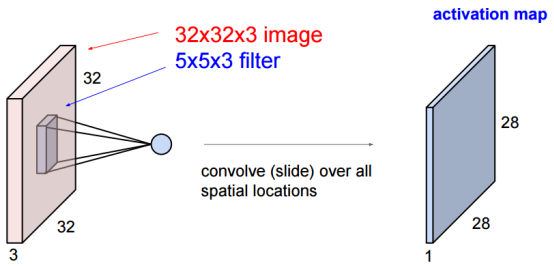

Figura 2.5: Muestra del funcionamiento de una capa convolucional [6].

Además, suelen incluir la capa de Reducción o *Pooling* que realiza la tarea de tomar una matriz de entrada y, como su propio nombre indica, hacer una reducción aplicando un filtro de nxn dimensiones, sacando así una nueva matriz. Entre las capas *Pooling* más usadas se encuentran las *Max Pooling* y las *Min Pooling*, las cuales de cada nxn elementos, selecciona el elemento de valor máximo o de valor mínimo, respectivamente.

Finalmente, la capa *Fully Connected (FC)* consiste en aplicar una operación matricial $Y = Wx+b$, donde x es el vector de entrada, W es la matriz de pesos respectiva, b es el vector de bias respectivo e Y es el vector de salida.

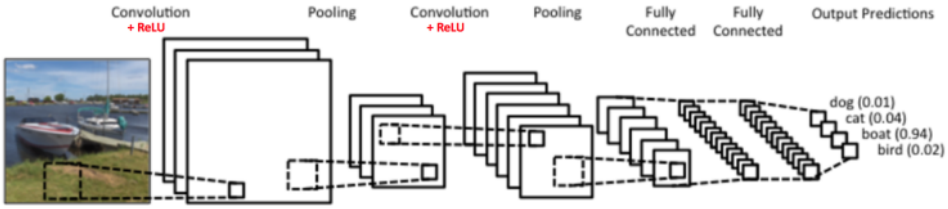

Figura 2.6: Ejemplo de una estructura de una red convolucional formada con dos capas convolucionales, dos capas de *Pooling* y dos capas *FC* [7].

2.2.3. Arquitecturas típicas para análisis de imágenes

A continuación hacemos un repaso de las arquitecturas que consideramos más relevantes a desde el primer uso existoso de CNN en *ImageNet*, bien por su importancia histórica o por las innovaciones que aportaron en el campo de la clasificación de imágenes. Como podemos ver en la figura 2.7, la cual representa el error de cada red obteniendo los cinco mejores resultados, se nota el gran salto que supuso la aparición de las CNN, ya que el error disminuyó casi a la mitad.

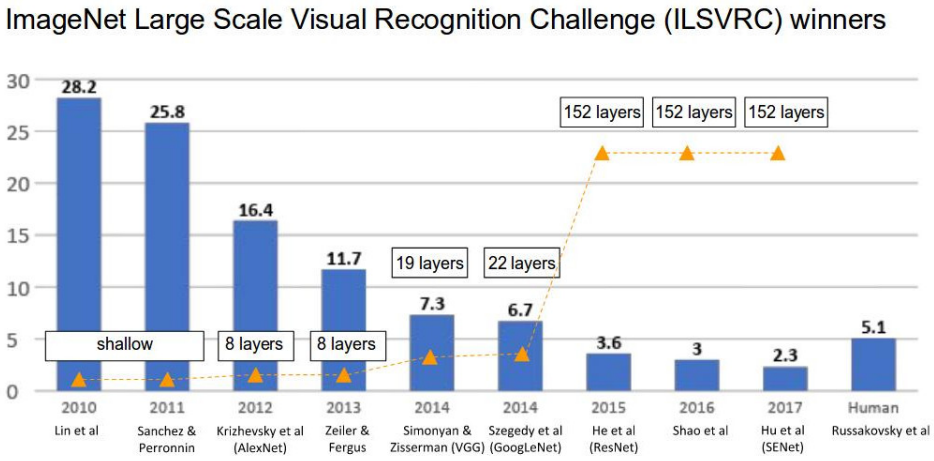

Figura 2.7: Ganadores del concurso ILSVRC desde el 2012 al 2017 [8].

AlexNet

Esta arquitectura fue creada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton [32]. Como dato a destacar de esta red, cabe decir que quedó la primera en el concurso ILSVRC-2012 consiguiendo un error del 16 % con los 5 resultados más probables y un 26 % de error con el resultado más probable.

Como podemos ver en la figura 2.8, consta de 5 capas convolucionales con filtros de tamaño 11x11, de las cuales 3 realizan *Max Pooling* posteriormente, y de 3 capas *FC*, por lo que tiene un total de 35.000 parámetros. Además esta red recibe una entrada de tamaño 224x224 y saca una salida de tamaño 1000x1 después de aplicar *softmax*, y está entrenada con la base de datos de *ImageNet* y con el uso de 2 GPUs para minimizar el error obtenido.

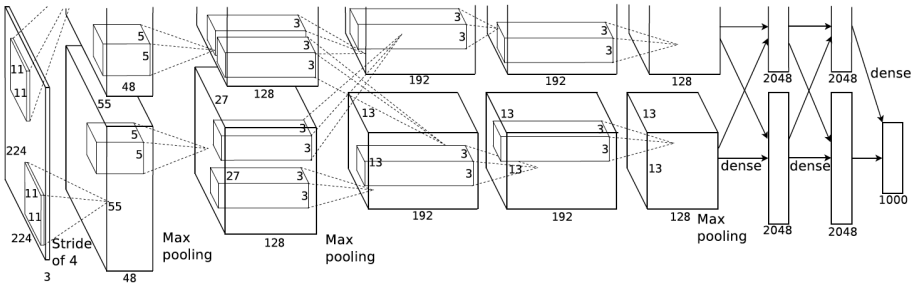

Figura 2.8: Arquitectura de AlexNet [9].

VGG

VGG fue creada or Karen Simonyan y Andrew Zisserman [33], y fue la subcampeona del concurso ILSVRC-2014.

Es una red similar a *AlexNet*, pero como podemos observar en la figura 2.9, VGG añade mayor profundidad, ya que está formada por un total de 16-19 capas entre capas convolucionales y capas *FC*. Además, las capas convolucionales aplican filtros de 3x3 y las capas de *Pooling* 2x2. Aunque el "defecto"que tiene esta red es el coste que tiene de su uso, ya que tiene un mayor número de parámetros y utiliza más memoria.

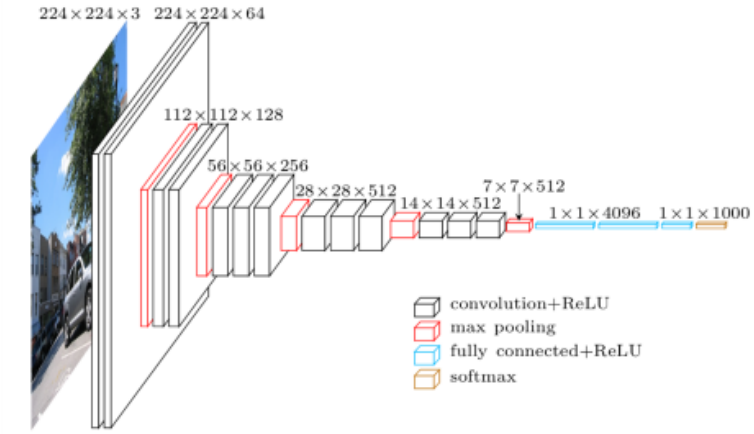

Figura 2.9: Arquitectura de VGG [10].

GoogLeNet

La empresa *Google* también tenía que sacar su propia arquitectura de redes neuronales, y entonces creó *GoogLeNet* [34], la cual ganó el concurso ILSVRC-2014.

Esta arquitectura está compuesta por 22 capas incluidas convolucionales, *Pooling* y una capa característica llamada *Inception*, como podemos ver en la figura 2.10. Esta última son un tipo de capa convolucional pero que se caracteriza por utilizar únicamente filtros 1x1, 3x3 y 5x5 de manera simultánea, por lo que de esta manera consigue disminuir enormemente el número de parámetros a calcular en comparación con *AlexNet*.

Esto lo consiguieron con lo que *Google* llamó *bottlenecks*, que se trataban de capas convolucionales con filtros 1x1 para así reducir la complejidad de la red. También incluyeron clasificadores

auxiliares, con el objetivo de ser más fácil la propagación de los gradientes hacia atrás y no suponga tanto coste. Así, además de reducir el número de parámetros y la complejidad, consiguieron como resultado un red más potente.

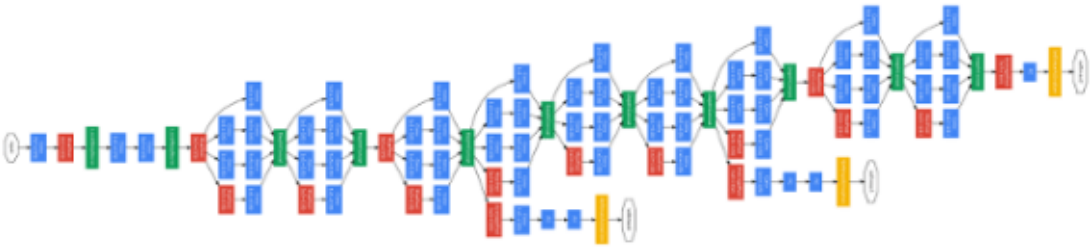

Figura 2.10: Arquitectura de GoogleNet [11].

ResNet

En los años siguientes, con la llegada de *ResNet* (*Residual Network*) [35] se produjo la llamada *Revolution of depth* [36], ya que esta arquitectura tenía un total de 152 capas, un número de capas bastante alto en comparación con la última arquitectura (*GoogleNet*) que tenía solo 22 capas.

ResNet fue desarrollada por Kaiming He et al. y fue la ganadora del ILSVRC-2015 obteniendo un 3.5 % de error dando los 5 resultados más probables. Se trata de una arquitectura con un diseño más simple que las anteriores, ya que prescinde del uso de capas *FC* y utiliza únicamente capas de convolución y *Max Pooling*, pero hay que destacar principalmente el uso de 152 capas.

Si *GoogleNet* ya tenía que utilizar clasificadores auxiliares para realizar la propagación de los gradientes... ¿Cómo lo hace *ResNet*? El "truco"que utiliza es la conexión de capas con las siguientes dejando una entre medias, así los saltos que tiene que realizar se reduce bastante en número y hace posible conseguir un error muy pequeño sin un algo coste. Podemos ver un ejemplo en la figura 2.11.

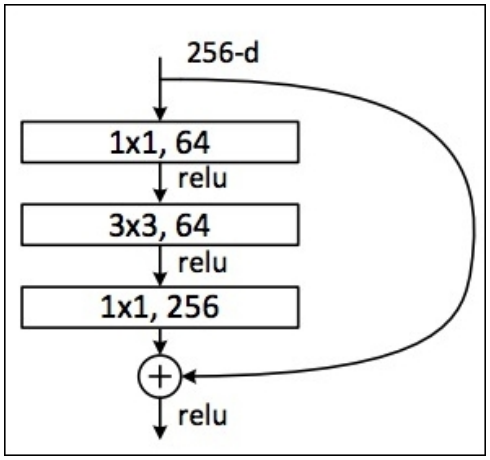

Figura 2.11: Muestra de los saltos que se realizan en la red *ResNet* [12].

Arquitecturas posteriores a *ResNet*

Finalmente, en los años 2016 y 2017 se han observado mejorías en el error pero no muy diferentes en comparación con *ResNet*, ya que han coonseguido bajarlo apenas un 1 % y utilizan el mismo número de capas, 152. Las ganadoras del concurso ILSVRC en los años 2016 y 2017 fueron *CUImage* y *SENet*, las cuales han sido mejoras de *ResNet*.

Cabe destacar en todo este proceso de las CNN que han conseguido obtener un error menor que el ser humano, ya que sabemos que cometemos un error del 5.1 % gracias al estudio que realizó Russakovsky [37], por lo que, una vez más, la máquina supera al hombre, en esta ocasión referente a la clasificación de imágenes.

2.3. Tensorflow

2.3.1. Herramientas SW

Actualmente existen numerosas herramientas software que nos permitan trabajar con *deep learning* y RNA. Entre los entornos más destacados encontramos *Caffe*, *Torch*, *Theano* y *Tensorflow*, y además cabe resaltar el uso de ciertos *wrappens* como *Keras*.

Caffe

Caffe (*Convolutional Architecture for Fast Feature Embedding*) [38] es una biblioteca de software libre creada por Yangqing Ja en la universidad de Berkeley para su uso con *deep learning*.

Se caracteriza principalmente por simplicidad a la hora de programar y su velocidad debido a su uso con tarjetas GPUs, lo que permite que sea muy útil en el mundo de la clasificación de imágenes.

Torch

Otra biblioteca bastante usada es *Torch* [39], una biblioteca de software libre, también desarrollada en la universidad de Berkeley y lanzada en 2002, basada en el lenguaje de programación Lua y que utiliza el lenguaje de script LuaJIT.

Se trata de una biblioteca muy popular sobre todo para su uso con RNA y optimización, y, al igual que Caffe, presenta facilidad a la hora de implementar código y es muy útil para el procesamiento de cualquier archivo audiovisual. Su uso con GPUs también le hace mucho más eficiente, sobre todo con arquitecturas de redes más complejas.

Theano

Una posible opción para trabajar con operaciones y algoritmos matemáticos es *Theano* [40], una biblioteca de *Python* creada en 2010.

Una de sus características principales es su gran integración con *NumPy*, otra biblioteca de *Python*, que nos permite la realización de grandes operaciones matriciales de una manera muy sencilla, y además le añade eficiencia y velocidad con el uso de GPUs.

Keras

Por último, queremos destacar una librería de *Python* que nos añade mucha flexibilidad de uso con el resto de herramientas software mencionadas, se trata de *Keras*. Es una librería de código abierto creada en 2015 por François Chollet, principalmente implementada para su uso con las RNA.

Como hemos mencionado antes, esta herramienta permite su adaptación con otras librerías como *Tensorflow* o *Theano* ya que tiene una estructura modular y muy extensible de código. Además permite su uso con CNNs y redes neuronales recurrentes, usando GPUs añadiéndole velocidad a su procesamiento, e incluso, una combinación de ambas, por lo que es bastante útil en el mundo del *deep learning*.

En nuestro caso, nos vamos a decantar por la elección de Tensorflow para el desarrollo de nuestro TFG.

2.3.2. Tensorflow: Qué es y cómo funciona

Tensorflow [14] lo definen en su página oficial de la siguiente forma: *"Es una librería de código libre para la realización de operaciones computacionales numéricas complejas"*. Por lo tanto, se trata de una librería bastante útil a la hora de implementar redes neuronales (especialmente redes convolucionales).

Fue desarrollada por personal del equipo *Google Brain*, pertenecientes al departamento de Inteligencia Artificial de Google.

Tiene una arquitectura flexible, lo que da pie a la facilidad de desarrollo de líneas de código a través de una gran variedad de plataformas. Además TensorFlow es un sistema de programación que nos permite representar los cálculos en forma de grafos.

Aquellos nodos donde guardamos las operaciones, las constantes o el valor de un resultado se llaman tensores. Y para calcular cualquier cosa dentro de TensorFlow, el grafo debe ser lanzado dentro de una sesión.

2.3.3. Redes neuronales con Tensorflow

Hay muchas posibles variaciones a la hora de construir una red neuronal con Tensorflow. Por tanto, vamos a mostrar un ejemplo ilustrativo para hacernos una idea: vamos a implementar una red básica **FC**, pero con la diferencia con respecto a la que mostramos en la figura 2.3 de no tener capa oculta:

```
import tensorflow as tf
import numpy as np

#vectores de entrada y de salida correcta
x_bottleneck = np.array([23,45,55,30,21])
corrects = np.array([0,0,0,1,0])

#vector de entrada con tamaño 1x5
bottleneck_input = tf.placeholder(
    tf.float32,
    shape=[None, 5],
    name='BottleneckInputPlaceholder')
```

```
#vector con clases correctas, total de 3 clases
ground_truth_input = tf.placeholder(tf.float32,
                                    [None, 3],
                                    name='GroundTruthInput')

#inicializamos la matriz de pesos siguiendo una distribucion normal con
# desviacion tipica 0.001
initial_value = tf.truncated_normal(
    [5,3], stddev=0.001)

#declaramos la matriz de pesos y de bias
layer_weights = tf.Variable(initial_value, name='final_weights')
layer_biases = tf.Variable(tf.zeros([3]), name='final_biases')

#realizamos la operacion Wx + b
logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases

#aplicamos softmax a la operacion anterior para obtener el vector salida
# y calculamos el coste
final_tensor = tf.nn.softmax(logits)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=ground_truth_input, logits=logits)

#calculamos el error medio aplicando la regularizacion L2 con constante
# de regularizacion 0.001
cross_entropy_mean = tf.reduce_mean(cross_entropy) +
                           tf.reduce_sum(layer_weights*layer_weights)*0.001

#descenso por gradiente para actualizar los parametros
train_step = tf.GradientDescentOptimizer(lr=0.01).minimize(cross_entropy_mean)

#creamos la sesion e inicializamos variables
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

#200 epocas de entrenamiento
for i in range(200):
  #seleccionamos un minibatch de 2 en cada epoca
  ind = np.random.permutation(range(5))[:2]
  input_images = x_bottleneck[ind]
  correct_predictions = corrects[ind]

  #ejecucíon de la red
  _,cost = sess.run([train_step, cross_entropy_mean],
                                          feed_dict={
                                            bottleneck_input: input_images,
                                            ground_truth_input: correct_predictions
                                            })
```

En el código anterior podemos observar como *Tensorflow* nos permite crear fácilmente una red neuronal a través de sencillas funciones.

Con la función *tf.Placeholder* creamos las variables que tomarán el valor que le asignemos. En este caso, lo utilizaremos para introducir los vectores de entrada y de salida correcta.

También utilizamos la función *tf.Variable*, la cual nos sirve para crear aquellas variables que queremos ir actualizando en cada ocasión que realicemos la ejecución de la red. Así será la decisión más oportuna para la matriz de pesos y de bias.

Además, *Tensorflow* nos permite la realización de operaciones matriciales, como también la aplicación de la función de activación que deseemos y el error obtenido. En el código anterior, podemos ver como ejemplo las funciones *tf.malmul*, *tf.nn.softmax* y *softmax_ cross_ entropy_ with_ logits*, respectivamente.

Cabe destacar el uso de un optimizador, el cual, indicándole la constante de aprendizaje, después de cada ejecución realiza automáticamente el cálculo de los gradientes y la actualización de los pesos y bias. En este código, hemos utilizado la función *tf.GradientDescentOptimizer*, a la cual le añadimos que queremos minimizar el error.

Por último, cabe destacar que para el funcionamiento de las operaciones de *Tensorflow*, necesitamos iniciar una sesión con *tf.Session*, inicializar las variables con *tf.global_ variables_ initializer* y la ejecución de la red con el función *run* de la sesión.

2.3.4. Ventajas y comparación con otras herramientas

Por lo que hemos visto en los apartados anteriores, *Tensorflow* parece ser una buena elección para la realización de RNA, ya que es una herramienta que se usa mucho en la actualidad, está muy bien documentado para poder utilizarlo siguiendo las guías que nos propone su página y además está actualizándose con frecuencia.

Además, tiene una gran compatibilidad con *Python* y el paquete *Numpy*, para la realización de todas las operaciones matriciales, o el uso de grafos para guardar toda la información de las variables y operaciones y tener la herramienta *Tensorboard* para poder plasmarlo gráficamente. Incluso cabe añadir el hecho de que *Tensorflow* no necesita calcular los gradientes de manera explícita, sino que lo calcula de manera automática, lo cual algo muy cómodo a la hora de implementar código.

También cabe destacar la velocidad de compilación que tiene. La mayoría de las anteriores características son propias también de *Theano*.

Como cosas negativas a destacar, aparece su lentitud de ejecución o su "tamaño.[en] comparación con otras herramientas como *Caffe* o *Torch*.

2.3.5. Modelos pre-entrenados

A día de hoy, entrenar un modelo desde cero que funcione bien y que nos permita obtener unos buenos resultados requeriría muchas horas de trabajo y el uso de máquinas muy potentes, además de la posesión de una gran base de datos. Por lo cual, está claro que estas condiciones no están al alcance de todos.

Entonces, para poder conseguir resultados, se acude a la utilización de un modelo ya entrenado anteriormente a partir de una base de datos. Éste se modifica para la base de datos y la clasificación que nosotros queremos realizar. Existen modelos pre-entrenados de la mayoría de las arquitecturas que ya hemos hablado en el apartado 2.2, pero nosotros vamos a utilizar las arquitecturas ***Inception V3*** y ***Mobilenet***.

Inception V3 es una arquitectura obtenida a partir de *GoogLeNet* con la especificación principal de introducir las capas *Inception* que mencionamos en apartados anteriores y fue entrenada con la base de datos de *ImageNet*.

Mobilenet es un modelo creado especialmente para su uso con *Tensorflow*, entrenado con la base de datos de *ImageNet*, tiene una estructura pequeña (comparada con *Inception* y su funcionamiento es más rápido y no exige demasiada potencia.

Entonces, visto que este procedimiento utiliza atributos generales que se pueden aplicar a la mayoría de las bases de datos, nos permite entrenar la última y obtener así una mayor eficacia, nos decantamos por él para realizar un clasificador potente que utilice una base de datos de flores.

3

Sistema, diseño y desarrollo

Para el desarrollo del código, hemos decidido utilizar *Python* y *Tensorflow*. Esto es debido a que, por un lado, *Python* es un lenguaje bastante intuitivo y muy fácil de utilizar a la hora de realizar operaciones computacionales de matrices y vectores. Y por otro lado, *Tensorflow* nos da toda la funcionalidad de redes neuronales y, específicamente, convolucionales; y además, nos permite realizar la fase de entrenamiento y de test de la red.

A continuación mostramos el desarrollo y el diseño del sistema realizado.

3.1. Preprocesado de imágenes

El primer paso que realizamos es preprocesar las imágenes convirtiéndolas a tamaño 32x32 y sacando su matriz de colores RGB mediante la adaptación de un script de Oxford [41] llamado *oxflower17.py*, que contiene una función llamada *load_data* que nos permite cargar todas las imágenes de dicha base de datos, redimensionarlas al tamaño nxn que deseamos y sacar la matriz con la información de todas las flores.

3.2. Redes propias

Luego, realizamos pruebas desarrolladas directamente en *Tensorflow* de diferentes CNN, en las cuales variamos los siguientes contenidos: incluyendo una o varias redes convolucionales, con filtro 7x7 y el número de capas de salida; añadiendo *Max Pooling* para reducir el tamaño de salida con filtro 2x2 y *step* 2; con o sin normalización de los valores obtenidos, ya sea con norma L1, L2 o *dropout*; y realizando una capa *Fully Connected* con los pesos y bias inicializados aleatoriamente siguiendo una distribución normal de media 0 y desviación típica 0'001. Para todas estas pruebas utilizamos la base de datos de flores *Oxford 17 Flowers*.

3.3. Redes pre-entrenadas

Al no obtener los resultados que deseábamos después de realizar muchas pruebas variando las arquitecturas y los parámetros, nos decantamos por realizar la búsqueda de una red preentrenada

y encontramos el script *retrain.py* [42]. Todas las pruebas que describimos más adelante se realizaron, primero, con la base de datos de flores *Oxford 17 Flowers* y, cuando conseguimos buenos resultados, pasamos a utilizar *Oxford 102 Flowers*.

Este script nos encajaba perfectamente con lo que buscábamos, ya que nuestro deseo era encontrar una red pre-entrenada que estuviese implementada con las librerías de *Python* y *Tensorflow*. Las imágenes que introducimos como base de datos son preprocesadas convirtiéndolas a un tamaño 224x224 y se obtiene su matriz de colores RGB como entrada a la red pre-entrenada. Este script nos permitía utilizar para entrenar dos arquitecturas, *Inception-V3* y *Mobilenet-1.0-224*, además de indicar el porcentaje de imágenes que utilizaría para entrenamiento o para test e, incluso, si querías utilizar un modelo ya implementado. En nuestro caso, los parámetros que utilizamos son: constante de aprendizaje 0.001, porcentaje de test y validación 0 % (ya que queríamos pasar todas las imágenes por la red pre-entrenada), 650 épocas y *batch* de 100 imágenes para cada época.

También, este script te permitía añadir tu red propia para la última capa para realizar el entrenamiento con nuestras bases de datos, pero como nos interesaba tenerlo en un fichero aparte para agilizar el periodo de pruebas, modificamos el script *retrain.py* y sacamos la última capa a un fichero aparte, llamado ***red.py***.

Ahora nuestro script *retrain.py* se encarga de coger la base de datos que le introduzcamos y devolvernos el vector de salida de la penúltima capa. Así, podemos introducirlo como entrada en nuestra última capa que, como hemos dicho antes, la hemos llevado al fichero *red.py*. Esta última capa consiste en una FC, que contiene una matriz de pesos, un bias y que aplica Softmax a la salida de la operación $y = Wx + b$ y calcula el error o coste con entropía cruzada. En esta red variaremos el valor de algunos parámetros como la constante de aprendizaje y la constante de regularización, y utilizará 2 posibles optimizadores, *Adam* y Gradiente por Momento con Nesterov, además de aplicar diferentes regularizaciones, L1, L2 y ambas a la vez.

El código desarrollado en el fichero *red.py* usando la arquitectura *Mobilenet-1.0-224* es el siguiente:

```
import tensorflow as tf
import numpy as np

x_bottleneck = np.load("bottleneck_102flowers_mobilenet_1.0_224.npy")
correct = np.load("labels_102flowers_mobilenet_1.0_224.npy")

bottleneck_input = tf.placeholder(
    tf.float32,
    shape=[None, 1001],
    name='BottleneckInputPlaceholder')

ground_truth_input = tf.placeholder(tf.float32,
                                    [None, 102],
                                    name='GroundTruthInput')

initial_value = tf.truncated_normal(
    [1001, 102], stddev=0.001)

layer_weights = tf.Variable(initial_value, name='final_weights')
layer_biases = tf.Variable(tf.zeros([102]), name='final_biases')

logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
```

```
final_tensor = tf.nn.softmax(logits)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=ground_truth_input, logits=logits)

correct_prediction = tf.equal(tf.argmax(final_tensor, 1),
tf.argmax(ground_truth_input, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

ind_train =[]
ind_test = []
tam = [40,60,40,56,65,45,40,85,46,45,87,87,49,48,49,41,85,82,49,56,40,59,
91,42,41,41,40,66,78,85,52,45,46,40,43,75,108,56,41,67,127,59,130,93,40,
196,67,71,49,92,258,85,93,61,71,109,67,114,67,109,50,55,54,52,102,61,42,
54,54,62,78,96,194,171,120,107,251,137,41,105,166,112,131,86,63,58,63,154,
184,82,76,66,46,162,128,91,66,82,63,49,58,48]

#distribucion aleatoria de los indices
ini = 0
for i in range(len(tam)):
  ind_perm = np.random.permutation(range(ini,ini+i))
  tr1 = int(0.5*i)
  tr2 = int(0.25*i)
  ind_train.extend(ind_perm[:tr1])
  ind_test.extend(ind_perm[tr1:tr1+tr2])
  ini = ini+i

#diferentes valores que toman los parametros
learning_rate = [1e-3,1e-2]#,1e-1]
N = 200
opti = ["Adam","Nesterov1", "Nesterov0.9", "Nesterov0.85"]
momentum = [0.85, 0.9, 1]
regularization = ["L2", "L1", "L1L2"]
reg = [0.001, 0.005, 0.01]

sess = tf.Session()

for lr in learning_rate:
  for opt in opti:
    if opt == "Adam":
      optimizer = tf.train.AdamOptimizer(lr)
    elif opt == "Nesterov0.8":
      optimizer = tf.train.MomentumOptimizer(learning_rate=lr,
       momentum=momentum[0], use_nesterov=True)
    elif opt == "Nesterov0.9":
      optimizer = tf.train.MomentumOptimizer(learning_rate=lr,
       momentum=momentum[1], use_nesterov=True)
    else:
      optimizer = tf.train.MomentumOptimizer(learning_rate=lr,
       momentum=momentum[2], use_nesterov=True)
```

```
for reg_cte in reg:
  for regu in regularization:
    if regu == "L1":
      cross_entropy_mean = tf.reduce_mean(cross_entropy) +
       tf.reduce_sum(np.abs(layer_weights))*reg_cte
    elif regu == "L2":
      cross_entropy_mean = tf.reduce_mean(cross_entropy) +
       tf.reduce_sum(layer_weights*layer_weights)*reg_cte
    else:
      cross_entropy_mean = tf.reduce_mean(cross_entropy) +
       tf.reduce_sum(np.abs(layer_weights))*reg_cte +
       tf.reduce_sum(layer_weights*layer_weights)*reg_cte

    train_step = optimizer.minimize(cross_entropy_mean)

    result_train = []
    result_test = []
    saver = tf.train.Saver({"final_weights": layer_weights,
     "final_biases": layer_biases})

    #cada combinacion la ejecutamos 10 veces para obtener un promedio de eficacia
    for no_use in range(10):
      init = tf.global_variables_initializer()
      sess.run(init)
      for i in range(N):
        input_images = x_bottleneck[ind_train]
        correct_predictions = correct[ind_train]
        # training
        train_accuracy,_,_ = sess.run([accuracy, train_step, cross_entropy_mean],
         feed_dict={
                                              bottleneck_input: input_images,
                                              ground_truth_input: correct_predictions
                                              })
        if i+1 == N:
        #guardamos el valor de las variables W y b
          saver.save(sess, "/tmp/model.ckpt")

          print("Case %d: training accuracy %g" % (no_use+1, train_accuracy))
          # test
          test_accuracy,prob_clasif, aciertos_fallos =
           sess.run([accuracy, final_tensor, correct_prediction],
           feed_dict={
               bottleneck_input: x_bottleneck[ind_test],
               ground_truth_input: correct[ind_test]
               })
          print("Validation accuracy: %g." % test_accuracy)

          result_train.append(train_accuracy)
          result_test.append(test_accuracy)

    # calculamos la eficacia media de training y test
```

```
train_accuracy_mean = np.mean(np.array(result_train))
train_accuracy_desv = np.std(np.array(result_train))
test_accuracy_mean = np.mean(np.array(result_test))
test_accuracy_desv = np.std(np.array(result_test))
```

Finalmente, encontramos una combinación de parámetros, optimizador y regularización que nos permitía obtener unos buenos resultados, por lo que al terminar el entrenamiento guardamos la matriz de pesos W y el vector bias b.

3.4. Modelo final

Una vez llegado a este punto, el último paso consistía en la creación de un nuevo script *test.py*, el cual consistiría en una red igual que la que implementamos en el fichero *red.py* que utilizase la matriz de pesos y bias guardados después de realizar nuestro entrenamiento y, utilizando como vector de entrada cualquier imagen (preprocesada por la red pre-entrenada), indicase las tres clases más probables en las que podría caracterizarse la flor de la imagen introducida, para así garantizar con una probabilidad muy alta que una de las tres categorías ofrecidas es la flor procesada.

4

Experimentos Realizados y Resultados

A continuación procedemos a describir las pruebas realizadas, especificando en cada caso:

- Arquitectura utilizada.
- Base de datos utilizada.
- Parámetros que vamos a variar, indicando el rango y los diferentes valores que vamos a probar.
- Resultado más sobresaliente y resultados generales obtenidos.
- Utilizaremos el 50 % de las imágenes de cada clase para la fase de entrenamiento, el 25 % para validación y el 25 % para la fase de test, todas ellas elegidas de manera aleatoria.

4.1. Redes propias

A continuación vamos a realizar pruebas a partir de redes desarrolladas desde cero con *Tensorflow* y *Python*.

4.1.1. Prueba 1 Red Neuronal

En esta primera red que implementamos, decidimos que tuviese la siguiente estructura: una capa convolucional, con filtro de 7x7, seguida de una capa *Max Pooling* con filtro 2x2 y paso 2, y de una capa FC. Finalmente aplicamos Softmax a la salida obtenida.

La base de datos que utilizaremos será *Oxford 17 Flowers*, y la constante de aprendizaje usará los valores 0.001 y 0.01.

Los resultados de eficacia de entrenamiento no superaron el 60 %.

4.1.2. Prueba 2 Red Neuronal

En la siguiente red que implementamos, decidimos que tuviese la siguiente estructura: dos capas convolucionales, ambas filtro de 3x3; realizando Max Pooling con filtro 2x2 y paso 2 detrás de cada una de ellas; una capa FC y aplicando Softmax.

La base de datos que utilizaremos será *Oxford 17 Flowers*, y la constante de aprendizaje usará los valores 0.001 y 0.01.

Los resultados de eficacia de entrenamiento no superaron el 60 %.

4.1.3. Prueba 3 Red Neuronal

En la última red que implementamos, decidimos que tuviese la siguiente estructura: tres capas convolucionales, todas con filtro de 3x3; realizamos Max Pooling con filtro 2x2 y paso 2 detrás de cada una de ellas; una capa FC y aplicando Softmax.

La base de datos que utilizaremos será *Oxford 17 Flowers*, y la constante de aprendizaje usará los valores 0.001 y 0.01.

α	0.001	0.01
Red 1	0.596	0.55
Red 2	0.543	0.51
Red 3	0.489	0.45

Cuadro 4.1: Tabla de los mejores resultados de test con cada red usando la base de datos *Oxford 17 Flowers*

Entonces, como podemos ver en la tabla 4.1, los resultados de eficacia de test no superaron el 60 %. Asi que pasamos a realizar pruebas con redes pre-entrenadas.

4.2. Red pre-entrenada

En todas las pruebas que mostramos a continuación, se han ejecutado con un total de 200 épocas y tendremos en cuenta los siguientes aspectos:

- La constante de aprendizaje varía entre los valores 0.001, 0.01 y 0.1.
- El optimizador utilizado será *Adam* o *Momentum Optimizer*, variando el valor del momento entre 0.9 y 1, y usando el modo Nesterov.
- La regularización que utilizaremos será L1, L2 o una fusión de ambas, y variando la constante de regularización (CR) entre los valores 0.001, 0.01 y 0.1.
- En cada posible combinación de parámetros, realizamos un total de 10 casos y promediamos las eficacias obtenidas para, así, obtener un resultados consistente.

4.2.1. Prueba 1 Red pre-entrenada

En esta prueba usaremos la base de datos *Oxford 17 Flowers* y la arquitectura *Inception V3*, realizando todas las posibles combinaciones de parámetros como indicamos anteriormente.

Los resultados de test obtenidos aparecen en las tablas 4.2, 4.3 y 4.4:

	CR	L1	L2	L1L2
	0.001	0.9469 ± 0.0046	**0.9603 ± 0.0028**	0.945 ± 0.003
Adam	0.005	0.9172 ± 0.0103	**0.9633 ± 0.0017**	0.9125 ± 0.0026
	0.01	0.8411 ± 0.0021	**0.9606 ± 0.0027**	0.8214 ± 0.0048
	0.001	0.9153 ± 0.0171	0.9228 ± 0.0181	0.915 ± 0.017
Nesterov0.9	0.005	0.8556 ± 0.0428	0.9281 ± 0.0319	0.8547 ± 0.0398
	0.01	0.7506 ± 0.0586	0.9283 ± 0.0316	0.7442 ± 0.0577
	0.001	0.9072 ± 0.0121	0.9314 ± 0.0104	0.9247 ± 0.0228
Nesterov1.0	0.005	0.9164 ± 0.0125	0.8864 ± 0.0695	0.9061 ± 0.0146
	0.01	0.8522 ± 0.0177	0.8572 ± 0.0953	0.8369 ± 0.0185

Cuadro 4.2: Tabla de resultados de test con arquitectura *Inception V3*, Constante de Aprendizaje 0.001 y la base de datos *Oxford 17 Flowers*

	CR	L1	L2	L1L2
	0.001	0.9319 ± 0.0107	**0.9631 ± 0.0025**	0.9461 ± 0.0031
Adam	0.005	0.9089 ± 0.0059	**0.9642 ± 0.0032**	0.9114 ± 0.004
	0.01	0.8308 ± 0.0083	0.9611 ± 0.0037	0.8264 ± 0.0036
	0.001	0.9383 ± 0.0077	0.9561 ± 0.0069	0.9396 ± 0.0023
Nesterov0.9	0.005	0.9083 ± 0.0097	**0.9631 ± 0.0025**	0.9047 ± 0.0086
	0.01	0.8328 ± 0.0139	0.9625 ± 0.0028	0.8067 ± 0.0107
	0.001	0.912 ± 0.0173	0.9058 ± 0.0286	0.9339 ± 0.0202
Nesterov1.0	0.005	0.9108 ± 0.0076	0.8889 ± 0.025	0.9072 ± 0.0093
	0.01	0.8428 ± 0.0158	0.9 ± 0.0299	0.8225 ± 0.0073

Cuadro 4.3: Tabla de resultados de test con arquitectura *Inception V3*, Constante de Aprendizaje 0.01 y la base de datos *Oxford 17 Flowers*

	CR	L1	L2	L1L2
	0.001	0.8017 ± 0.193	0.9492 ± 0.0423	0.8483 ± 0.01871
Adam	0.005	0.7372 ± 0.1436	0.8681 ± 0.0877	0.52 ± 0.1005
	0.01	0.6358 ± 0.1366	0.7889 ± 0.1598	0.4178 ± 0.1055
	0.001	0.9483 ± 0.0018	**0.9628 ± 0.0025**	0.9467 ± 0.008
Nesterov0.9	0.005	0.9069 ± 0.0041	**0.965 ± 0.0014**	0.8572 ± 0.0014
	0.01	0.7261 ± 0.007	0.9069 ± 0.0	0.6881 ± 0.0096
	0.001	0.9339 ± 0.0114	0.9256 ± 0.0116	0.9464 ± 0.0054
Nesterov1.0	0.005	0.9089 ± 0.0094	0.9536 ± 0.0259	0.8664 ± 0.0046
	0.01	0.7358 ± 0.0101	**0.9556 ± 0.0072**	0.6983 ± 0.0181

Cuadro 4.4: Tabla de resultados de test con arquitectura *Inception V3*, Constante de Aprendizaje 0.1 y la base de datos *Oxford 17 Flowers*

Como podemos observar en las tablas 4.2, 4.3 y 4.4, podemos destacar la obtención general de un resultado eficiente, superando el 90 % de efectividad, y siempre obteniendo los mejores resultados usando una regularización L2 con CR menor y usando el optimizador Adam. El mejor resultado obtenido ha sido un 96.5 % usando Nesterov con momento 0.9, regularización L2 con CR 0.005 y constante de aprendizaje 0.1.

El siguiente paso consiste en realizar las mismas pruebas con la otra arquitectura de la que disponemos.

4.2.2. Prueba 2 Red pre-entrenada

En esta ocasión utilizaremos la base de datos *Oxford 17 Flowers* y la arquitectura *MobileNet 1.0 224*, realizando todas las posibles combinaciones de parámetros como indicamos anteriormente.

Los resultados de test obtenidos aparecen en las tablas 4.5, 4.6 y 4.7:

	CR	L1	L2	L1L2
Adam	0.001	0.9772 ± 0.0021	**0.9975 ± 0.0044**	0.9778 ± 0.0025
	0.005	0.9678 ± 0.005	**0.9886 ± 0.0008**	0.9719 ± 0.0015
	0.01	0.9692 ± 0.0032	0.985 ± 0.0025	0.9725 ± 0.0008
Nesterov0.9	0.001	0.9805 ± 0.0018	0.9869 ± 0.0028	0.9795 ± 0.0014
	0.005	0.9756 ± 0.003	0.9875 ± 0.0026	0.9756 ± 0.003
	0.01	0.9725 ± 0.0008	0.9872 ± 0.0025	0.9728 ± 0.0017
Nesterov1.0	0.001	0.9708 ± 0.0176	**0.9906 ± 0.0053**	0.9633 ± 0.0222
	0.005	0.9636 ± 0.0132	0.9422 ± 0.0604	0.9675 ± 0.0068
	0.01	0.9706 ± 0.0085	0.95 ± 0.0403	0.9681 ± 0.0052

Cuadro 4.5: Tabla de resultados de test con arquitectura *MobileNet 1.0 224*, Constante de Aprendizaje 0.001 y la base de datos *Oxford 17 Flowers*

	CR	L1	L2	L1L2
Adam	0.001	0.9708 ± 0.0043	0.9875 ± 0.0094	0.9736 ± 0.0033
	0.005	0.9678 ± 0.0031	0.9819 ± 0.0043	0.9714 ± 0.0028
	0.01	0.9708 ± 0.0042	0.9803 ± 0.0015	0.9725 ± 0.0038
Nesterov0.9	0.001	0.9781 ± 0.0015	**0.9947 ± 0.006**	0.9778 ± 0.0
	0.005	0.9669 ± 0.0032	**0.9892 ± 0.0031**	0.9722 ± 0.0
	0.01	0.9683 ± 0.0025	**0.9881 ± 0.0025**	0.9722 ± 0.0
Nesterov1.0	0.001	0.9453 ± 0.0175	0.9592 ± 0.0294	0.9272 ± 0.017
	0.005	0.9678 ± 0.0086	0.9419 ± 0.0178	0.9694 ± 0.0056
	0.01	0.9708 ± 0.0042	0.9428 ± 0.0154	0.9714 ± 0.0058

Cuadro 4.6: Tabla de resultados de test con arquitectura *MobileNet 1.0 224*, Constante de Aprendizaje 0.01 y la base de datos *Oxford 17 Flowers*

	CR	L1	L2	L1L2
	0.001	0.9578 ± 0.0052	0.9692 ± 0.0148	0.9339 ± 0.0561
Adam	0.005	0.9117 ± 0.0977	0.9778 ± 0.0025	0.8578 ± 0.1278
	0.01	0.8769 ± 0.0943	0.9769 ± 0.0025	0.8431 ± 0.0997
	0.001	0.9664 ± 0.0139	**0.9947 ± 0.0082**	0.9131 ± 0.1294
Nesterov0.9	0.005	0.8733 ± 0.1698	**0.9811 ± 0.008**	0.9164 ± 0.0924
	0.01	0.9125 ± 0.1084	**0.9802 ± 0.0008**	0.8275 ± 0.1209
	0.001	0.93 ± 0.0282	0.9094 ± 0.0244	0.8261 ± 0.0918
Nesterov1.0	0.005	0.9017± 0.0358	0.9194 ± 0.0399	0.8867 ± 0.0527
	0.01	0.8606 ± 0.005	0.8847 ± 0.0542	0.8275 ± 0.0704

Cuadro 4.7: Tabla de resultados de test con arquitectura *MobileNet 1.0 224*, Constante de Aprendizaje 0.1 y la base de datos *Oxford 17 Flowers*

Como podemos ver ahora en las tablas 4.5, 4.6 y 4.7, podemos observar que obtenemos mejores resultados que con la arquitectura anterior, llegando incluso a rozar el 100 % de eficacia, y vuelven a lograrse usando los mismos parámetros que en la prueba anterior. El mejor resultado obtenido ha sido un 99.75 % usando Adam, regularización L2 con CR 0.001 y constante de aprendizaje 0.001.

También llama la atención que según aumentamos la constante de aprendizaje, el optimizador Nesterov tiende a funcionar mejor que Adam.

Por lo que, ya que con la misma base de datos y utilizando esta arquitectura hemos obtenido mejores resultados, procedemos a realizar la siguiente prueba con esta misma arquitectura y una base de datos más amplia.

4.2.3. Prueba 3 Red pre-entrenada

En esta ocasión utilizaremos la base de datos *Oxford 102 Flowers* y la arquitectura *MobileNet 1.0 224*, realizando todas las posibles combinaciones de parámetros como indicamos anteriormente.

Los resultados de test obtenidos se muestran en las tablas 4.8, 4.9 y **??**:

	CR	L1	L2	L1L2
	0.001	0.9207 ± 0.0013	**0.9304 ± 0.0005**	0.9201 ± 0.0015
Adam	0.005	0.8941 ± 0.002	**0.9302 ± 0.0006**	0.8935 ± 0.0014
	0.01	0.8402 ± 0.0016	**0.929 ± 0.0005**	0.836 ± 0.0016
	0.001	0.9048 ± 0.0008	0.9109 ± 0.0016	0.9053 ± 0.0009
Nesterov0.9	0.005	0.8671 ± 0.0008	0.9118 ± 0.0012	0.8676 ± 0.0008
	0.01	0.7774 ± 0.0011	0.9115 ± 0.0011	0.7754 ± 0.0006
	0.001	0.9011 ± 0.0008	0.8968 ± 0.0005	0.9014 ± 0.0012
Nesterov1.0	0.005	0.8861 ± 0.0024	0.8969 ± 0.0008	0.8857 ± 0.0017
	0.01	0.847 ± 0.0038	0.8979 ± 0.0008	0.8468 ± 0.0031

Cuadro 4.8: Tabla de resultados de test con arquitectura *MobileNet 1.0 224*, Constante de Aprendizaje 0.001 y la base de datos *Oxford 102 Flowers*

	CR	L1	L2	L1L2
	0.001	0.9043 ± 0.0025	0.9152 ± 0.0038	0.9073 ± 0.0032
Adam	0.005	0.8786 ± 0.0029	**0.929 ± 0.0009**	0.8872 ± 0.0032
	0.01	0.8334 ± 0.003	**0.9283 ± 0.0007**	0.8452 ± 0.0041
	0.001	0.9186 ± 0.0005	0.9267 ± 0.0011	0.9187 ± 0.001
Nesterov0.9	0.005	0.8907 ± 0.0005	**0.9282 ± 0.001**	0.8901 ± 0.001
	0.01	0.8322 ± 0.0011	0.9278 ± 0.0006	0.8332 ± 0.0011
	0.001	0.8864 ± 0.0026	0.9043 ± 0.0008	0.8815 ± 0.0015
Nesterov1.0	0.005	0.8794 ± 0.004	0.9034 ± 0.0013	0.8874 ± 0.0033
	0.01	0.8106 ± 0.0052	0.895 ± 0.0016	0.8365 ± 0.0044

Cuadro 4.9: Tabla de resultados de test con arquitectura *MobileNet 1.0 224*, Constante de Aprendizaje 0.01 y la base de datos *Oxford 102 Flowers*

Como se puede observar en este caso, sólamente realizamos esta vez pruebas con LR = 0.001 y LR = 0.01, ya que en los casos anteriores obtuvimos los peores resultados con LR = 0.1. Ahora, fijándonos en los mejores resultados obtenidos en las tablas 4.8 y 4.9, éstos rondan el 90 % con los mismos parámetros que en las ocasiones anteriores, destacando en la cima el resultado de **93.04 %**.

Así, una vez terminada toda la batería de pruebas, podemos llegar a la conclusión de que nuestra red más eficiente consta de las siguientes características:

- Arquitectura: **MobileNet 1.0 224**
- Optimizador: **Adam**
- Constante de Aprendizaje: **0.001**
- Regularización: **L2**
- Constante de Regularización: **0.001**

En resumen, hemos conseguido adaptar una red pre-entrenada para nuestra herramienta de clasificación de flores usando una base de datos que contiene 102 categorías diferentes y tiene una eficacia del 93.04 %, por lo que esto cumple con satisfacción nuestro objetivo del 80-90 % que teníamos planteado.

5

Conclusiones y trabajo futuro

5.1. Conclusiones

Una vez realizada toda la parte teórica del documento, toda la parte práctica comprendida por la batería de pruebas realizadas y nuestro análisis final, podemos sacar las siguientes conclusiones:

- Las Redes Convolucionales son la herramienta más usada actualmente para el reconocimiento de imágenes.
- El uso de una red preeentrenada permite conseguir resultados mucho más eficientes que con una red propia, ya que no tenemos los sistemas y herramientas necesarios para conseguir dichos resultados.
- Entrenar una red con una base de datos más amplia te permite obtener mejores resultados, ya que tienes una mayor generalización.

5.2. Trabajo Futuro

La idea inicial del trabajo futuro que tendría esta herramienta sería el crear una aplicación móvil con soporte para Android o IOS, o adaptar dicha herramienta con soporte online para poder utilizarla en cualquier navegador web. En ambos casos, la idea es que fuese de uso libre para cualquier usuario que quiera o necesite la identificación de una flor.

Dicha aplicación permitiría la entrada de una imagen de una flor por cámara o foto guardada, la procesaría y devolvería las 3 posibles flores candidatas con mayor probabilidad de ser la flor que pregunta, adjuntando a cada una de ellas tanto su nombre como una imagen para poder identificarla.

Por tanto, puede ser útil para un usuario que simplemente quiera utilizarlo mientras da un paseo por el campo como por un usuario que necesite saber la identidad de una flor en su entorno de trabajo (botánica, jardinería, farmacia...).

Además, otra posibilidad que podemos valorar es la idea de extender la base de datos para así conseguir una mejora de nuestras redes. Por ejemplo, sería una buena idea añadir imágenes

de flores propias de la Península Ibérica para obtener una base de datos que podamos utilizar con más fiabilidad en España. Y una vez hecho esto, faltaría realizar un entrenamiento completo con toda la base de datos ampliada antes de ponerla en producción.

Glosario de acrónimos

- **RNA**: Redes Neuronales Artificiales
- **CNN**: Redes Neuronales Convolucionales
- **FC**: Fully Connected
- **TFG**: Trabajo de Fin de Grado
- **BBDD**: Bases de Datos
- **BoW**: Bag of Words
- **ILSVRC**: ImageNet Large Scale Visual Recognition Challenge
- **GD**: Descenso por Gradiente

Bibliografía

[1] Ejemplo de flores similares entre si. *https://es.dreamstime.com/fotos-de-archivo-libres-de-regal*

[2] Maria-Elena Nilsback and Andrew Zisserman. Oxford 102 flowers. http://www.robots.ox.ac.uk/ vgg/data/flowers/102/, *page Apartado: .ºxford 102 Flowers", 2009.*

[3] Maria-Elena Nilsback and Andrew Zisserman. Oxford 17 flowers. http://www.robots.ox.ac.uk/ vgg/data/flowers/17/, *page Apartado: .ºxford 17 Flowers", 2009.*

[4] Image semantic segmentation using fully convolutional neural network. https://nrupatunga.github.io/2016/06/01/image-semantic-segmentation-using-fully-convolutional-neural-network/.

[5] The prediction of undersaturated crude oil viscosity: An artificial neural network and fuzzy model approach. https://www.researchgate.net/figure/Back-propagation-multilayer-ANN-with-one-hidden-layer$_fig2_241741756$.

[6] Muestra de funcionamiento de la capa convolucional. https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolutional$_neural_networks.html$

[7] An intuitive explanation of convolutional neural networks. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

[8] Imagenet large scale visual recognition challenge (ilsvrc) winners. http://cs231n.stanford.edu/slides/2018/cs231n$_2018_lecture09.pdf$.

[9] Understanding alexnet. https://sushscience.wordpress.com/2016/12/04/understanding-alexnet/.

[10] Leaf app: Leaf recognition with deep convolutional neural networks. https://www.researchgate.net/figure/VGG16-architecture-16$_fig2_321829624$.

[11] How to calculate the number of layer for googlenet. https://stackoverflow.com/questions/47633393/how-to-calculate-the-number-of-layer-for-google.

[12] Residual networks (resnet). https://www.safaribooksonline.com/library/view/building-machine-learning/9781786466587/ch08s05.html.

[13] Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/.

[14] Tensorflow. Tensorflow. https://www.tensorflow.org, *page Apartado: .About Tensorflow", 2016.*

[15] Chih-Fong Tsai. Bag-of-words representation in image annotation: A review. https://doi.org/10.5402/2012/376804, *page 19, 2012.*

[16] David G. Lowe. Object recognition from local scale-invariant features. http://www.cs.ubc.ca/ lowe/papers/iccv99.pdf*, page 8, 1999.*

[17] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). http://www.cs.zju.edu.cn/ gpan/course/materials/SURF.pdf*, page 14, 2008.*

[18] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: an efficient alternative to sift or surf. http://www.willowgarage.com/sites/default/files/orb$_f$*inal.pdf*, *page* 8, 2011.

[19] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. https://www.edwardrosten.com/work/rosten$_2$006$_m$*achine.pdf*, *page* 14, 2006.

[20] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. https://www.cs.ubc.ca/ lowe/525/papers/calonder$_e$*ccv*10.*pdf*, *page* 8, 2010.

[21] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. https://arxiv.org/pdf/1506.01497.pdf*, page 14, 2016.*

[22] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. http://yann.lecun.com/exdb/publis/pdf/farabet-pami-13.pdf*, page 15, 2013.*

[23] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. https://arxiv.org/pdf/1312.4659.pdf*, page 9, 2014.*

[24] Yann LeCun. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/*, page Apartado: "The MNIST Database of Handwritten Digits", 2012.*

[25] Alex Krizhevsky. The cifar-10 dataset. http://www.cs.utoronto.ca/"Librerias de Machine Learning", 2017.

[26] Standford University. Imagenet. *http://www.image-net.org/*, page Apartado: ÏMAGENET", 2016.

[27] Imagenet large scale visual recognition challenge. *http://image-net.org/challenges/LSVRC/.*

[28] Jia Deng, Wei Dong, Richard Socher, Lia-Jia Li, Kai Ki, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *https://www-cs.stanford.edu/groups/vision/pdf/ImageNet$_C$VPR*2009.*pdf*, *page*$Apartado$: "$Introduction$", 2009.

[29] Ieee computer society conference on computer vision and pattern recognition. *https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=5206873*, page 36, 2009.

[30] How to retrain an image classifier for new categories. *https://www.tensorflow.org/tutorials/image$_r$etraining.*

[31] Y. Liu, F. Tang, D. Zhou, Y. Meng, and W. Dong. Flower classification via convolutional neural network. *https://ieeexplore.ieee.org/document/7818296/*, 2016.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf3*, page 9, 2012.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *https://arxiv.org/pdf/1409.1556.pdf*, page 14, 2014.

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *https://arxiv.org/pdf/1409.4842.pdf*, page 12, 2014.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *https://arxiv.org/pdf/1512.03385.pdf*, page 12, 2015.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning. $http://image\text{-}net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf, page$ 36, 2015.

[37] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *https://arxiv.org/pdf/1409.0575.pdf*, page 36, 2015.

[38] Berkeley. Caffe. *http://caffe.berkeleyvision.org/*, pages–, -.

[39] Berkeley. Torch. *http://torch.ch/*, pages–, 2002.

[40] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *http://deeplearning.net/software/theano/*, pages–, 2012.

[41] Deep learning library featuring a higher-level api for tensorflow. *https://github.com/tflearn/tflearn/blob/master/tflearn/datasets/oxflower17.py*.

[42] How to retrain an image classifier for new categories. $https://www.tensorflow.org/tutorials/image_retraining$.